# SRI International

## TerraVision: A Terrain Visualization System

Technical Note No. 540

April 22, 1994

By: Yvan G. Leclerc, Senior Computer Scientist
Steven Q. Lau Jr., Software Engineer

**Approved for Public Release; Distribution Unlimited**

**This work was supported in part by the Advanced Research Projects Agency under Contract F19628-92-C-0071.**

**The views, opinions and/or conclusions contained in this note are those of the author and should not be interpreted as representative of the official positions, decisions, or policies, either expressed or implied, of the Advanced Research Projects Agency, or of the United Stated Government.**

333 Ravenswood Avenue - Menlo Park, CA 94025-3493 - (415) 326-6200 - FAX: (415) 326-5512 - Telex: 334486

DRAFT

# TerraVision: A Terrain Visualization System[1]

**Yvan G. Leclerc and Stephen Q. Lau Jr.**
**AI Center**
**SRI International**

## 1.0 Introduction

TerraVision is a system for visualizing terrain. That is, it allows a user to view, in real time, a synthetic recreation of a real landscape created from elevation data and a large number of aerial images of that landscape. The goal of the system is to allow a user to roam about a terrain comprising tens to hundreds of gigabytes of data at arbitrary speeds and from arbitrary vantage points. This is distinctly different from a flight simulator, in which the goal is to accurately simulate the controls and limitations of a particular aircraft. Here, the goal is to allow the user to be free of such limitations so that he/she may roam about the terrain at will. Of course, the limitations of a particular aircraft could be imposed if one wanted to build a simulator based on TerraVision technology.

The difficulty in creating an ideal terrain visualization system can be seen by considering a few typical scenarios.

Consider the case when the user wishes to get an overview of the entire terrain. That would be done by viewing the terrain from a sufficiently high altitude that the entire terrain would be visible on the screen. Since the entire terrain is visible, a straightforward implementation would require accessing the entire terrain database to create a single image! This is clearly not acceptable. What is needed is to be able to access a small, coarse resolution, version of the terrain database when creating such a view.

Conversely, if the user is looking down on the terrain from a low altitude, only a small portion of the terrain is visible, but a high resolution version of the data is required for creating the view.

A more complex situation occurs when the user is near ground level and looking out towards the horizon. Here, a very large fraction of the terrain can be visible in the distance because of perspective. Moreover, one needs high resolution data in the foreground and low resolution data in the background.

---

1. TerraVision was created as the first application of the Multidimensional Applications Gigabit Internet Consortium (MAGIC) high-speed network testbed, funded in part by ARPA contract F19628-92-C-0071. Yvan Leclerc can be reached at leclerc@ai.sri.com, and Stephen Lau at lau@ai.sri.com.

Another difficulty with ground-level views is that, as the user turns his/her head, or travels quickly over the terrain, the amount of new terrain visible per second is potentially enormous.

The above examples suggest some of the basic problems that an ideal terrain visualization system must solve:

1. How can a synthetic view of large areas of the terrain be created without accessing large amounts of data at a time and without requiring inordinate amounts of computation time to determine what parts of the terrain are visible?
2. How can one efficiently determine, access, and use the appropriate resolution data for arbitrary viewpoints?
3. How can one minimize the amount of new data required per second as the user moves over the terrain?
4. How can a real-time representation of the terrain be created in the face of uncertain delays in database retrieval and the possibility of lost data?

The solutions adopted by TerraVision are described in detail in the following sections. The basic idea is to create a multi-resolution representation of the terrain data (both the elevation and image data), divided into equal-sized elements called tiles. This way, only those tiles of the resolution required for a given viewpoint need to be retrieved from the database. Because of inherent delays in retrieving tiles, TerraVision requests tiles well in advance of their being visible by predicting the user's future viewpoint, expanding this viewpoint to take into account the uncertainty in the prediction, and requesting those tiles in the future viewpoint that are not currently in memory. Keeping a coarse resolution representation of the entire database in memory, plus a large cache of previously requested tiles, allows TerraVision to render a view of the terrain at all times. Furthermore, the specific search and pre-fetching algorithms described below allow TerraVision to adapt itself to the actual rate at which tiles can be retrieved from the database, as well as to the actual rate at which they can be rendered.

The distinct advantage of the above approach is that the terrain database can be remote from the graphics station used to render the synthetic views. This is a critical ability if the user wishes to visualize a database that is much larger than the local storage capacity of the graphics station. As indicated above, TerraVision basically uses an incremental retrieval of the database as required by the user, rather than forcing the user to copy a part of the database to local storage, visualizing that part, and repeating this until he/she has found the portion of the terrain that was of interest. Indeed, the user may not know which portion of the terrain is truly of interest until he/she has seen it from just the right point of view!

An additional advantage of directly using the remote database, rather than a local copy, is that the database can be kept up to date without concern for updating local copies. With appropriate representations of the changes made to the database, it is possible for the user not only to visualize the current state of the terrain, but also to visualize the changes that have taken place.

# 2.0 TerraVision Search and Rendering Algorithms

TerraVision has several components. One of the most important of these is determining which portions of the terrain are visible from a given viewpoint, and what resolution should be used for rendering each portion. In the following sections, we define the basic terrain rendering problem, describe some of the issues that need to be resolved, and finally describe the algorithm used by TerraVision for rendering large terrains.

## 2.1 Terrain Rendering Problem Definition

Abstractly, the terrain is defined as elevation and color as a function of $(x, y)$ coordinates on a plane; the elevation function is called $z(x, y)$, and the color function is a vector-valued function called $\grave{c}(x, y)$. In the simplest case, the components of the color function are the red, green, and blue intensities at a given point. In general, other material properties, such as specularity and transparency can be included.

It is possible to generalize the formulation presented here by specifying that elevation and color are functions of some other two-dimensional $(u, v)$ coordinate system, such as the Universal Transverse Mercator (UTM) coordinate system or the polar coordinates of a sphere. In this case, the resulting surface defined by the set of points $(u, v, z(u, v))$ needs to be transformed to its cartesian $(x, y, z)$ coordinates before rendering. However, the filtering, sampling, and search algorithms described below can be used as long as the transformation is locally approximately linear.

There are two basic classes of algorithms that can be used to render terrain. The first is ray-tracing, in which one or more rays are cast from the focal point of the camera through each pixel in the view window. Each ray intersects $z(x, y)$ at zero or more points. The $(x, y)$ coordinate of the closest intersection point is then used as the index into the color function, thereby specifying the color of that ray. When there is more than one ray per pixel, a weighted average of the colors of the rays is used as the color of that pixel. This algorithm is a kind of an inverse of the way images are formed in real cameras, where light rays reflect from a surface and are “accumulated’ via the lens of the camera.

The second class of algorithms is z-buffering. A simple form of this algorithm is to sample the $(x, y)$ plane on a square grid, forming a set of square facets with vertices $(x_i, y_j, z(x_i, y_j))$. The vertices of each facet can then be projected onto the view plane, forming a trapezoid in the viewing window. For each pixel within the trapezoid, one can determine the distance to the elevation surface by bilinearly interpolating the distances at the four vertices. This distance can then be used to compute the $(x, y)$ coordinate of the intersection point of the ray with the surface. As with the ray-tracing algorithm, this coordinate is used as the index into the color function, and a weighted average of the colors can be used if more than one point is used per pixel.

Both of the algorithms, as described above, suffer from the effects of perspective.

In the case of the ray-tracing algorithm, this takes the form of severe aliasing where the surface is far from the camera. This is because the rays from adjacent pixels intersect the

surface at increasingly large intervals in the $(x, y)$ plane as the surface recedes from the camera. Consequently, if the number of rays per pixel is fixed, eventually the sampling interval in the $(x, y)$ plane falls below $1/2\omega_0$, where $\omega_0$ is the bandwidth of one or the other of the terrain functions, and aliasing occurs. A possible solution to this problem is to increase the number of rays cast per pixel as a function of the distance to the elevation surface, but this eventually becomes too expensive.

In the case of the z-buffering algorithm, this takes the form of intolerable rendering time. The reason is that, if the $(x, y)$ plane is sampled uniformly at intervals of at least $1/2\omega_0$, the number of facets that need to be rendered per pixel increases as the surface recedes from the camera, eventually becoming so large as to make it impossible to render in a given amount of time. A possible solution is to increase the sampling interval as a function of distance, but then aliasing occurs.

A second kind of problem also needs to be addressed if a large area, say 40 kilometers on a side, is to be rendered at high resolution, say 32 meter intervals for elevation and 1 meter intervals for color, resulting in approximately 1.5 million elevation samples and 1.5 billion color samples. Namely, one needs to able to determine which parts of the terrain are visible for any given viewpoint in a computationally efficient manner. This is a problem for both the ray-tracing and z-buffering algorithms.

In the remainder of Section 2.0, we describe a solution to both the problems of aliasing and of search size for the case of the z-buffering algorithm. However, much of what is described below can also be used for ray-tracing algorithms.

The basic problem, then, is to create the highest quality rendering of the terrain using a z-buffering algorithm, subject to such limitations as:

1. The maximum number of facets that can be rendered per second, which is a function of the area of the facets on the screen.
2. The maximum number of texture-maps that can be used per second. (A texture-map is an array of color values.)
3. The maximum number of texture-map pixels that can be rendered per second.
4. The maximum rate at which the data can be brought into main memory.
5. The desired frame-rate, which may be a function of velocity. For example, when moving quickly, one requires a high frame-rate to minimize lag and temporal aliasing, but one can afford to use a spatially cruder rendering because people are less sensitive to detail in moving imagery. Conversely, when standing still, one can afford to spend more time rendering.

The approach used here is to sample the terrain functions in a non-uniform fashion such that the distance between neighboring samples at a given point, as projected onto the view window, is roughly uniform across the screen. Consequently, the number of polygons per frame and the number of texture-map pixels per frame remains roughly constant, independent of the distance to the surface, and rendering time becomes roughly constant for each frame.

For a given point on the terrain and a given viewpoint, let's call the distance between neighboring samples, as measured in the $(x, y)$ plane, the "terrain sampling interval" at that point, and the distance between the projection of neighboring samples, as measured on the screen in pixels, the "screen sampling interval" at that point. Let $s_z(x, y)$ and $s_c(x, y)$ denote the terrain sampling interval for the elevation and color functions, respectively. Let $r(x, y, s(x, y))$ denote the corresponding screen sampling interval for a given sampling interval function.

By specifying that the screen resolution should be approximately constant over the visible portion of the terrain, and what that resolution should be for the elevation and color functions, one can control both the quality of the rendering and the amount of data required for the rendering. Thus, the two parameters that TerraVision controls as a function of graphics engine rendering performance, data retrieval bandwidth, viewpoint, velocity, and other considerations, are the screen resolutions of the elevation and color functions, denoted $r_z$ and $r_c$, respectively.

## 2.2 Non-uniform Sampling Implies Non-uniform Filtering

Before describing the mechanism by which the non-uniform sampling rate is computed in an efficient manner, it is important to realize that a non-uniform sampling rate implies a non-uniform filtering of the functions $z(x, y)$ and $c(x, y)$. Otherwise, as indicated earlier, aliasing effects will become apparent.

How can one perform a non-uniform filtering of a function in a computationally efficient manner? First, let's pose the problem more formally. Given a band-limited function $f(x, y)$ of bandwidth $\omega_0$, what is the largest sampling interval $s_0$ such that the function can be exactly reconstructed? The answer, according to the sampling theorem, is $s_0 = 1/2\omega_0$. Conversely, if we wish to sample the function at given a sampling interval $s$ without aliasing, it must first be bandlimited to a maximum bandwidth of $\omega = 1/2s$ by convolution with an ideal low-pass filter of that bandwidth. Let's call this ideally low-pass-filtered function $f(x, y, l(s))$ for a given sampling interval $s$, where $l(s) = \log s/s_0$, is called the sampling level.[1] (It is more convenient to index the function by the sampling level $l(s)$ instead of the sampling interval $s$, as we shall see shortly.) Note that, by construction, $f(x, y, l(s))$ can be exactly represented by a set of discrete samples on a square grid:

$$(x_i, y_j) = (x_0 + is, y_0 + js)$$

by using sync function interpolation. One can approximate $f(x, y, l(s))$ by using other interpolation functions, such as bilinear interpolation, on the discrete samples, though a higher sampling rate is generally required to ensure a good approximation. Let's call the bilinearly-interpolated approximation function $f_b(x, y, l(s))$.

1. All logarithms in this paper are base 2.

The above states that one can use bilinear interpolation on a square grid of discrete samples to approximate a low-pass-filtered function. But if we were to use this approach for arbitrary sampling intervals $s$, one would need an infinite number of such sets of discrete samples. An efficient alternative is to compute $f_b(x, y, l_k)$ for a discrete set of sampling intervals $s_k$, where $l_k = l(s_k)$, and then to interpolate between the pair of approximation functions $f_b(x, y, l_k)$ and $f_b(x, y, l_{k+1})$ when $l_k \le l(s) < l_{k+1}$.

For example, suppose that $s_k = s_0 2^k$. That is, the sampling interval doubles from one set to the next, starting at $s_0 = 1/(2\omega_0)$, where $\omega_0$ is the bandwidth of the original function $f(x, y)$. Then, one can approximate $f_b(x, y, l(s))$ as follows:

$$f_b(x, y, l(s)) \approx (1 - r) f_b(x, y, l) + r f_b(x, y, l + 1)$$

where

$$l = \lfloor l(s) \rfloor$$
$$r = l(s) - \lfloor l(s) \rfloor$$

That is, the integer part of $l(s)$ defines the pair of bandlimited functions to use, while the fractional part defines their weights. Let's call this trilinearly interpolated approximation function $f_t(x, y, l(s))$.

The above suggests that one can approximate the non-uniformly sampled function $f(x, y, l(s(x, y)))$ by the trilinearly interpolated approximation function $f_t(x, y, l(s(x, y)))$ using a pyramid of sampled functions $f_b(x_i, y_j, l_k)$, where $s_k = s_0 2^k$ and $l_k = l(s_k) = k$. Each level of the pyramid requires only one quarter the number of samples of the previous level, so that the total number of samples for the entire pyramid is only four thirds that of the first level. Such an approximation is not only computationally efficient, but also efficient in its use of memory.

## 2.3 Computing Screen Resolution for Non-Uniform Sampling

For a given sampling interval $s$, screen resolution was defined above as the distance in pixels between the projection of neighboring samples of the surface. Since the distance between the projection of neighboring points $(x_i, y_j, z(x_i, y_j))$ and $(x_i + s, y_j, z(x_i + s, y_j))$ is generally different from that between neighboring points $(x_i, y_j, z(x_i, y_j))$ and $(x_i, y_j + s, z(x_i, y_j + s))$ due to foreshortening, we need to define a non-directional screen resolution function. One such function is the minimum of these two distances. Using the minimum of the two distances has the property of reducing the number of facets required for the rendering, but at the expense of image quality. Another possi-

bility is the maximum of the two distances, which has the opposite property. Let $r(x, y, s)$ denote the desired function of these two distances, for a given sampling interval $s$. For convenience, let $r(x, y) = r(x, y, 1)$.

When the sampling interval $s(x, y)$ is small relative to the distance between the surface and the camera at all points $(x, y)$ of interest, then $r(x, y, s(x, y))$ can be approximated as

$$r(x, y, s(x, y)) \approx s(x, y)\, r(x, y)$$

Consequently, if we wish to specify that the screen resolution for $z(x, y)$, say, should be a constant value $r_z$ (for example, we may wish to specify that the $r_z = 10$, resulting in surface facets that are approximately 10 pixels on a side, for a total of approximately 10,000 polygons for a 1000 by 1000 pixel window), then the sampling interval for elevation should vary as

$$s_z(x, y) = \frac{r_z}{r(x, y)}$$

The above appears well-defined, but isn't quite, since $r(x, y)$ depends on the elevation function $z(x, y)$. But recall that, if we are to avoid aliasing problems, the value of the elevation function depends on the rate at which we wish to sample the function. That is, it depends on $s_z(x, y)$! Rather than attempting to solve the recursive relationship defined above, we simply specify that the elevation function used in computing $r(x, y)$ be $z(x, y, l(s))$ for some fixed, pre-determined value of $s$. In other words, we will use a relatively smooth version of $z(x, y)$ to computed the desired sampling rate. This has the advantage that the sampling rate will vary relatively slowly over the surface. Furthermore, one can accurately approximate $s_z(x, y)$ by sampling $r(x, y)$ at intervals of length $s$, storing these values in an array, and using bilinear interpolation. This will be useful in defining computationally efficient means for sampling the surface, as we shall see below. The disadvantage is that screen resolution won't be quite as constant as if we were to solve exactly for the recursively defined sampling rate. We would expect that the largest errors in using the above approximation are when the surface is closest to the camera, but this is also presumably where we are forced to use the highest-resolution data available in any case. Thus, the errors in computing sampling rate in the above fashion are not important.

Note that, in general, we need to compute a different sampling interval function $s_c(x, y)$ for the color function $c(x, y)$ (for example, we may wish to specify that $r_c = 1$, so that samples of the color function project at approximately 1 pixel intervals on the screen). To do this efficiently, recall that it is the logarithmic function $l(s_z(x, y))$

$$l(s_z(x, y)) = l_z(x, y) = \log r_z - \log(s_0 r(x, y))$$

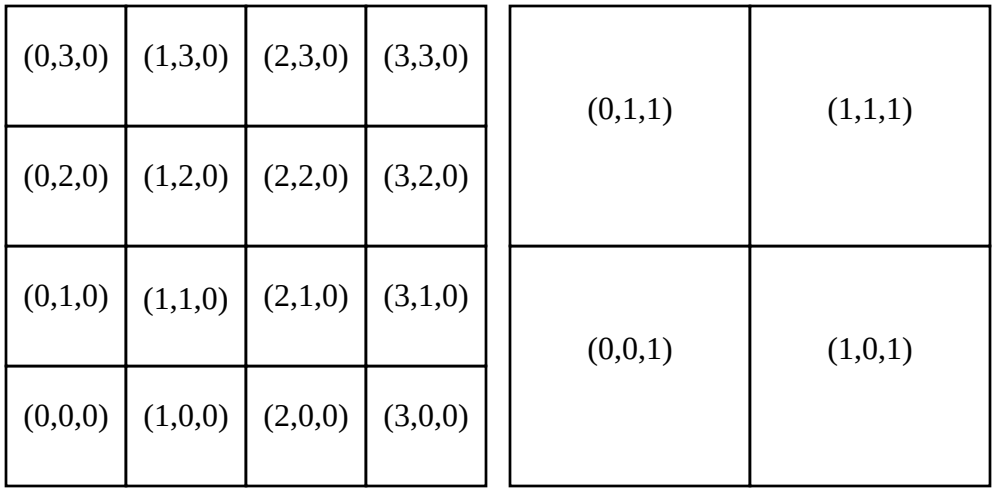


**FIGURE 1. Two levels of the quad-tree. Each node of the quad-tree corresponds to a volume of space. The square area of the node is defined as the union of the open set of the square and all points along the left and bottom boundaries, excluding the topmost and rightmost points on the boundaries, respectively. Level 0 corresponds to the smallest allowed division of the terrain.**

that is used to determine the level and weights for the interpolation of the functions in the pyramid. Thus, one need only compute $-\log(s_0 r(x,y))$ once, and simply add $\log r_z$ or $\log r_c$ when needed. For simplicity, a single value for $s_0$, equal to the smaller of $1/2\omega_0$ for the two functions, is used for the calculation of the level functions. This allows the use of a single indexing function for both elevation and color.

## 2.4 Computing Surface Visibility and Resolution

The above analysis described how to compute terrain resolution at any given point to achieve a desired screen resolution. In this section, we describe how to determine the visibility and resolution of the surface in a computationally efficient manner.

Our approach is to use a coarse-to-fine search on a quad-tree representation of the terrain. Each node of the quad-tree represents a volume in space defined by a square area in the $(x, y)$ plane, and the minimum and maximum elevation within that area. For simplicity, we call the length of the side of a node in $(x, y)$ coordinates the "size" of the node. The four children of a node correspond to a subdivision of the square area into four parts, as illustrated in Figure 1.

In a nut-shell, the search algorithm is as follows. The volume represented by the top-level node is projected onto the viewing plane. If no part of the projected volume lies within the view window, then the elevation surface within the node is definitely not visible. If some part of the volume lies within the view window, however, then the elevation surface is potentially visible. When this is so, a test is applied to determine whether or not the node should be sub-divided into its four children. If so, the search is carried on. Otherwise, it is stopped, and that portion of the terrain can be rendered. This recursive subdivision yields a

structure such as the one in Figure 2. (See below for details on the sub-division test and the resolution at which the terrain is rendered.)

Each node in the quad-tree also represents a square grid of samples of the terrain functions, called elevation and color tiles. If we fix the number of samples per tile at all levels (say 32 by 32 for elevation tiles and 128 by 128 for color tiles), then each level of the quad-tree represents a level of a resolution pyramid. This is because the size of a node doubles from one level in the quad-tree to the next, while the number of samples remains constant. Therefore, the interval between samples doubles from one level to the next in the quad-tree, just as the interval between samples doubles in the resolution pyramid. As we shall see below, this relationship between the quad-tree and the resolution pyramids is the key for rendering the terrain within a fixed amount of time, independent of view point.

A node in the quad-tree is identified by its position and level, denoted $(m, n, q)$. For example, in Figure 1 the children of node (1,1,1) are (2,2,0), (3,2,0), (2,3,0), and (3,3,0). Following are some assumptions about, and general formulas for, the names, positions, and sizes of nodes in the quad-tree.

1. The four children of node $(m, n, q)$ are $(2m, 2n, q-1)$, $(2m, 2n+1, q-1)$, $(2m+1, 2n+1, q-1)$, and $(2m+1, 2n, q-1)$.
2. The size of a node (i.e., the length of one of its side) at level 0 is $b$ meters.
3. The length of the side of a node at level $q$ is $b2^q$ meters.
4. For all values of $q$, the lower left-hand corner of nodes $(0, 0, q)$ is at $(x, y)$ position (0,0).
5. From items 3. and 4., the position of the lower-left-hand corner of node $(m, n, q)$ in $(x, y)$ coordinates is $\left( mb2^q, nb2^q \right)$.
6. The number of levels in the quad-tree is $= \lceil \log L/b \rceil$, where $L$ is the larger of the lengths of the terrain in the $x$ or $y$ directions.

Some assumptions required to maintain a simple relationship between nodes in the quad-tree and levels in the resolution pyramids are as follows. Note that $q$, the level number in the quad-tree, is not the same as the resolution number in the resolution pyramids. Item 6 below defines the relationship between the two.

**FIGURE 2. Leaf nodes of the terrain.**

1. The number of samples per node is a constant, and must be the square of a power of two for both elevation and color (possibly a different value); call the square root of the number of samples $n_z$ and $n_c$, respectively.
2. The sampling interval must double for both elevation and color from one level in the quad-tree to the next (this is a consequence of keeping the number of samples constant while doubling the length of a side of a node from one level to the next).
3. The smallest sampling interval for either elevation or color is denoted $s_0$. For example, if the highest resolution elevation data is at 32 meter intervals, and the highest resolution color data is at 1 meter intervals, then $s_0 = 1$ meter. It is important that $s_0$ be sufficiently small that a bilinear interpolation is a good approximation to the original elevation and color functions.
4. The sampling interval for both elevation and color must be powers of two times $s_0$.
5. For a given sampling interval $s$, the level number in both the elevation and color resolution pyramids is defined as $l(s) = \log s/s_0$. Continuing with the example above, the highest resolution elevation data would therefore be defined as level $\log 32/1 = 5$, while the highest resolution color data would be defined as level $\log 1/1 = 0$.
6. Let $q$ denote the level number of the quad-tree node. Then let $s_z(q)$ denote the sampling interval for the elevation data, and $s_c(q)$ denote the sampling interval for the color data. Note that $s_z(0) = b/n_z$, $s_c(0) = b/n_c$, $s_z(q) = s_z(0)\,2^q$, and $s_c(q) = s_c(0)\,2^q$. Also, let $l_z(q) = \log s_z(q)/s_0 = q + \log b/(s_0 n_z)$ and $l_c(q) = \log s_c(q)/s_0 = q + \log b/(s_0 n_c)$ denote the elevation and color sampling levels, respectively.

Ignoring color data for the moment, the search for which component of the terrain is potentially visible and what resolution it should be rendered at is carried out by traversing the quad-tree as follows. The search begins with a list of nodes to be tested (initially just node $(0, 0, N)$). If that node is potentially visible (as defined earlier), then a test is carried out to determine whether or not the node should be subdivided. If the test is positive, the four children of the node are added to the bottom of the list of nodes to be examined. If not, this node is marked as a “leaf” node for the rendering of elevation data.

The subdivision test for a node at level $q$ depends on the sampling level function $l_z(x, y)$ defined earlier. If the sampling level for level $q$ of the quad-tree, $l_z(q)$ is larger than the smallest value of $l_z(x, y)$ over all samples $(x_i, y_j)$ within the node, then the node should be sub-divided into its four children. This ensures that the constant sampling interval within the tile is adequate to represent the non-uniform sampling interval function. We can reduce computation time by taking advantage of the slowly varying nature of the screen resolution function, and comparing $l_z(x, y)$ against $l_z(q)$ only at the four corners of the node, as illustrated in Figure 3.

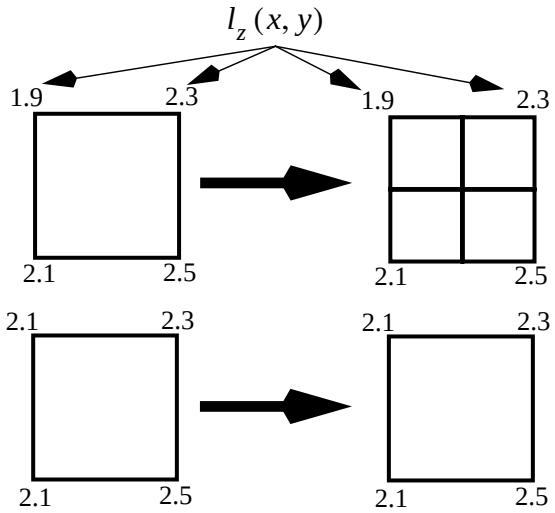


**FIGURE 3. Illustration of node subdivision. In the top row, the level 2 node is subdivided into four level 1 nodes because $l_z(x, y)$ is less than 2 at one of the corners. In the bottom row, the node does not need to be subdivided, since $l_z(x, y)$ is greater than 2 at all four corners.**

To render the leaf nodes produced as above, one needs to know the elevation value for each elevation sample, and one needs to know what resolution color data to use for texture mapping. The resolution of the color data is determined by further sub-dividing the quad-tree in a fashion similar to what was described above. However, it is no longer necessary to test for visibility, and it is $l_c(x, y)$ that is compared against $l_c(q)$ at the four corners of the node. Once a leaf node for the color data has been found (i.e., a node that can no longer be sub-divided because some of the color data at the next level is not currently in memory, or it fails the subdivision test), the texture map corresponding to the color leaf node is bound, and the elevation data covering the color leaf node can now be rendered.

## 2.5 Computation of Elevation

The elevation values for rendering node $(m, n, q)$ should, in principle, simply be the tri-linearly interpolated values

$$z_t(x_i, y_j, l_z(x_i, y_j)) = (1 - r)\, z_b(x_i, y_j, l) + r z_b(x_i, y_j, l + 1)$$

where

$$l = \lfloor l_z(x_i, y_j) \rfloor$$
$$r = l_z(x_i, y_j) - \lfloor l_z(x_i, y_j) \rfloor$$

for sample coordinates $x_i = mb2^l + is_z(q)$ and $y_j = nb2^l + js_z(q)$. However, a problem arises at the boundary between elevation tiles at different levels, as illustrated in Figure 4. The thick lines indicate the outline of a pair of adjacent tiles at levels $q$ and

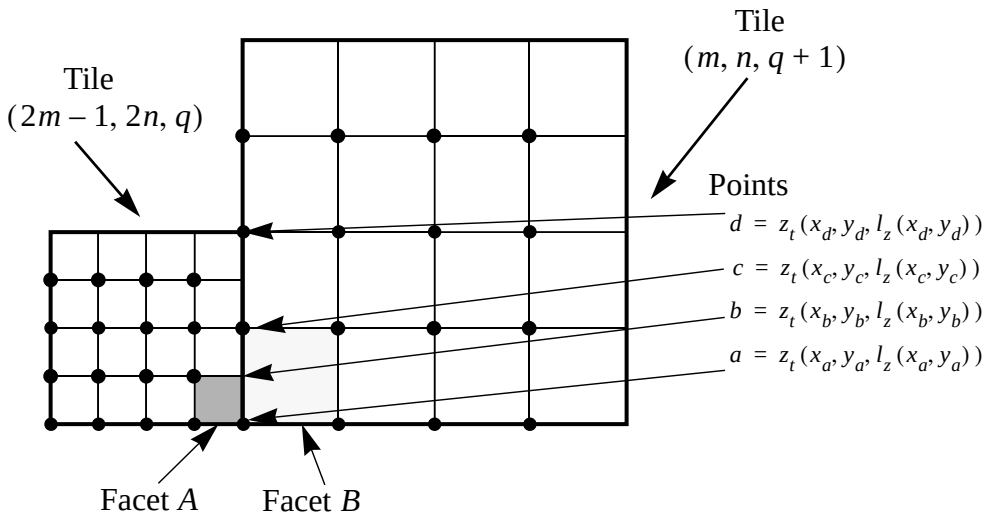


**FIGURE 4. Boundary between elevation tiles at two different levels. An elevation tile (and the corresponding quad-tree node) is defined to occupy a square area of the $(x, y)$ plane, with a closed boundary along the left and bottom sides, excluding the topmost point along the left side and the right-most point along the bottom side. With this definition, every point $(x, y)$ is covered by exactly one tile at any given level, namely tile $(\lfloor x/b \rfloor, \lfloor y/b \rfloor)$ for a tile of dimension $b$.**

$q + 1$, and the black discs indicate the positions of the samples within the tiles. In this illustration each elevation tile has sixteen samples ($n_z = 4$), and is rendered using sixteen bilinearly interpolated square facets. A property of bilinearly interpolated facets such as these is that elevation varies in a linear fashion along the $x$ and $y$ axes within the facet. Consequently, when rendered within facet $B$, point $b$ has an elevation that is the average of the elevations at points $a$ and $c$. However, in a straightforward implementation, the elevation of point $b$ when rendered withing facet $A$ would be directly evaluated as $z_t(x_b, y_b, l_z(x_b, y_b))$, since it is a vertex of that facet. In general, the elevations within facets $A$ and $B$ will be different at point $b$, giving rise to a "tear" in the surface along the boundary between the two facets. (Note that the elevations would be the same, however, if $l_z(x, y)$ were the same at all three points, and if it were greater than or equal to $l_z(q + 1)$. Given the subdivision rule defined above, the latter condition is always satisfied.)

There are two possible solutions to this problem. The first is to ignore it, under the assumption that the tears will be sufficiently small that they will be invisible. This is not necessarily a bad assumption, since the error in elevation at point $b$ is

$$\Delta(z_t(x_c, y_c, l_z(q + 1)) - z_t(x_a, y_a, l_z(q + 1))) +$$
$$\frac{\Delta}{2}(z_t(x_d, y_d, l_z(q + 2)) - z_t(x_a, y_a, l_z(q + 2)))$$

where

$$\Delta = l_z(x_c, y_c) - l_z(x_a, y_a)$$

is the change in $l_z(x, y)$ per sample interval at level $q + 1$. This assumes that $l_z(x, y)$ varies in a linear fashion within a tile, and that the sampling level changes by one from one tile to the next. This will be the case if $l_z(x, y)$ is computed using the bilinearly interpolated elevation function $z_b(x, y, l_z(q))$ for a sufficiently large value of $q$. In this case, not only will $l_z(x, y)$ vary linearly within a tile, but $\Delta$ should also be quite small. Nonetheless, it may be the case that the tears are visible for tiles that are near the viewer.

If tears along the boundaries are intolerable, it is necessary to treat the boundary points in a special fashion. The goal is to ensure that the elevation along the entire boundary of adjacent tiles be the same. Assuming that the quad-tree level can differ by no more than one for a pair of adjacent tiles, then the trilinearly interpolated elevation $z_t(x_i, y_j, l_z(x_i, y_j))$ can be used as the elevation at all of the interior vertices, as well as the even-numbered vertices along the boundary of a tile. It is only when the quad-tree level differs by one for a pair of adjacent tiles that a difference in elevation can occur, and this difference occurs only at the odd-numbered vertices along the boundary of the higher resolution tile, as at point $b$ of facet $A$ in the example of Figure 4.[1] At these vertices, the elevation should be the average of the two adjacent even-numbered vertices along the boundary. For example, if the odd-numbered vertex is $(x_i, y_j)$ at a horizontal boundary, the elevation at that vertex should be

$$\frac{z_t(x_{i-1}, y_j, l_z(x_{i-1}, y_j)) + z_t(x_{i+1}, y_j, l_z(x_{i+1}, y_j))}{2}$$

Unfortunately, there is no test that one can apply at vertex $(x_i, y_j)$, based on the level of the current tile $q$, the sampling level number of the current tile $l_z(q)$, and the sampling level function $l_z(x_i, y_j)$, that can unambiguously determine whether or not the adjacent tile is at a coarser resolution than the current one. This is because the sampling level of the adjacent tile depends on the value of the sampling level function at the four corners of the tile, as illustrated in Figure 5. Nonetheless, for a tile at level $q$, it is possible to determine with certainty when the neighboring tile is at the same or smaller level as the current one, namely, when $l_z(x_i, y_j) \leq q + 1$. When this condition is not satisfied, then it may be the case that the neighboring tile is at a coarser resolution. Thus, to ensure that no tear can occur along the tile boundary, the average elevation, as defined above, should be used at that point whenever $l_z(x_i, y_j) > q + 1$. The advantage of this rule is that it is local and very simple, yet it ensures that no tears can occur along tile boundaries. The disadvantage is that the surface will be rendered more coarsely than it should along tile boundaries in those cases when the neighboring tile is actually at the same level as the current one.

1. By odd-numbered vertices, I mean those vertices $(x_i, y_j)$ for which $i$ is odd along the top and bottom horizontal boundaries of the tile, and for which $j$ is odd along the left and right vertical boundaries of the tile.

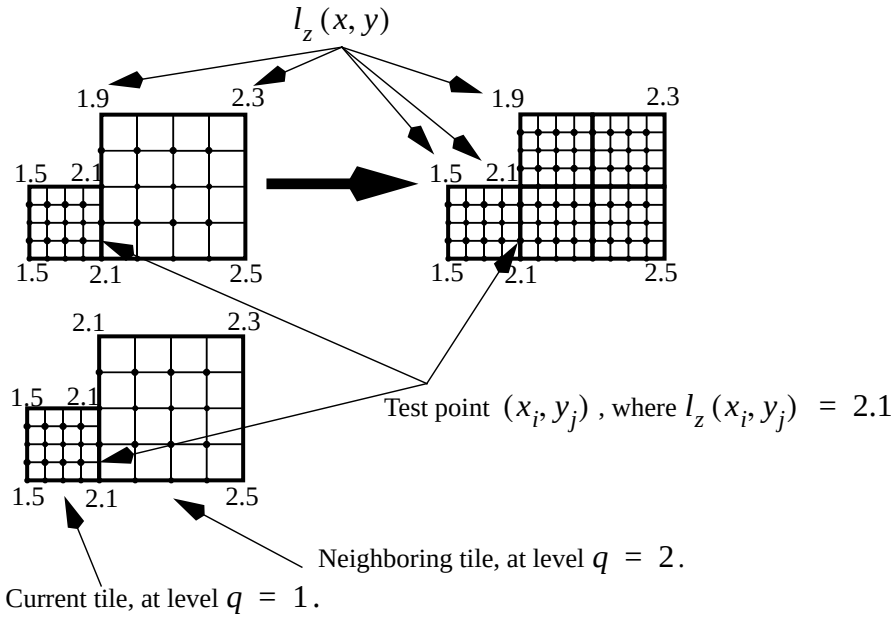


**FIGURE 5. Example of the inability to determine the quad-tree level of a neighbouring node at a given point $(x_i, y_j)$ on the basis of purely local information. In the top row, a value of $l_z(x_i, y_j) = 1.9$ in the upper left-hand corner of the larger tile causes it to be sub-divided, since the smallest value of $l_z(x, y)$ within the tile is less than 2. In the bottom row, the larger tile remains undivided because the smallest value of $l_z(x, y)$ is now greater than 2.**

Given the nature of $l_z(x, y)$, situations such as the one depicted in the top row of Figure 5 should happen rarely enough that the resulting artifact should be almost unnoticable.

## 2.6 Pipelining of Search and Rendering Algorithms

Given a graphics station with an architecture similar to the that of the popular Silicon-Graphics workstations, an efficient “unit of rendering” is a single color tile and an array of elevation samples, with associated data such as the $(x, y)$ coordinate of element $(0, 0)$ of the array, the start and end indices into the array, and the distance between samples. The single color tile can then be placed into texture-map memory (special memory that is part of the graphics hardware), and the elevation values can be quickly converted into quad-strips and transmitted to the hardware graphics pipeline. To keep the hardware pipeline as busy as possible, the process that converts the elevation data into quad-strips must be as simple as possible, and should never be idle. To this end, the rendering component of Ter-raVision has been divided into three pipelined stages, as illustrated in Figure 6. Each stage is implemented as a separate process, communicating via shared memory. $r_z$ and $r_c$

The first stage in the pipeline is the “Truncated Quad Tree Generator”. This stage traverses the terrain quad tree in the manner described in Section 2.4, and creates a “truncated quad tree” data structure. A truncated quad tree is one in which the nodes correspond to the nodes of the terrain quad tree, but only visible nodes are included, and only those nodes that have been subdivided have children. In addition, leaf nodes for elevation and color are

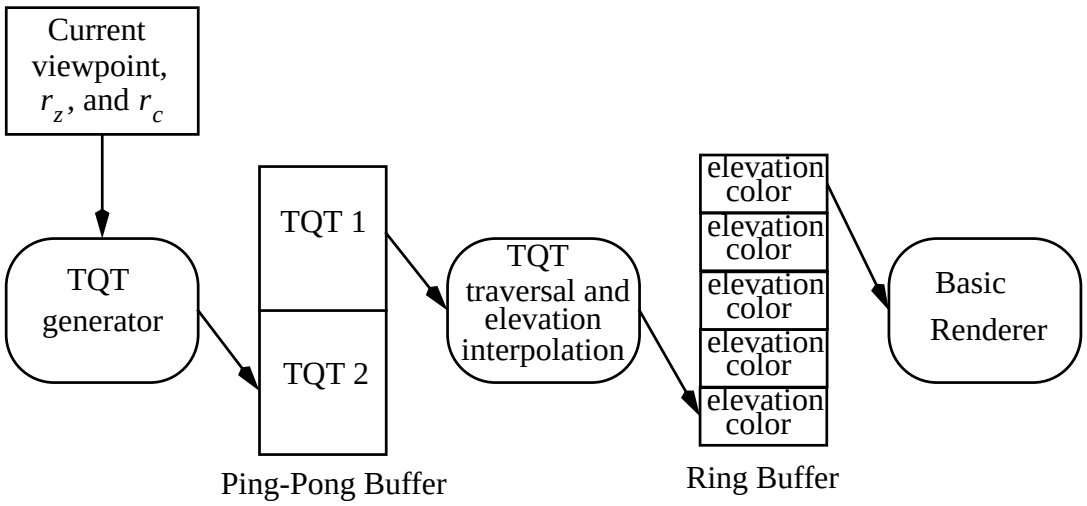


**FIGURE 6. TerraVision rendering pipeline.**

so marked. This data structure is placed in one half of a ping-pong buffer in shared memory.

The next stage traverses the truncated quad tree in the other half of the ping-pong buffer, processes elevation and color data at leaf nodes in preparation for rendering by the “Basic Rendering” stage, and places the processed data onto the ring buffer. Before the truncated quad tree is traversed, a special marker is placed in the ring buffer indicating to the Basic Renderer that a new frame has been started. To understand the manner in which the truncated quad tree is traversed, let’s first consider the situation in which all the necessary color tiles (i.e., all of the color leaf nodes) are currently in memory. The tree is traversed in a depth-first fashion until a node marked as an elevation leaf node is encountered. (Because of the manner in which the tree is generated, it is impossible to encounter a color leaf node before encountering an elevation leaf node.) When an elevation leaf node of level $q$ is encountered, the data in the elevation tiles at levels $l_z(q)$, $l_z(q+1)$ (and possibly $l_z(q+2)$) are used to compute trilinearly interpolated elevation samples, as described in Section 2.5. These values are stored in an array of size $(n_z+1) \times (n_z+1)$. The traversal of the tree is continued in a depth-first fashion until color leaf nodes are encountered. For each color leaf node encountered, a pointer to the corresponding color tile and to the previously computed elevation array is placed in the first available slot of the ring buffer. When all color leaf nodes for the current elevation node are exhausted, the search for the next elevation leaf node is continued. Note that when the ring buffer is sufficiently long, it may be necessary to use more than one elevation array because there may be more slots in the ring buffer than there are color leaf nodes for a given elevation leaf node. (In fact, it is possible, but unlikely, that as many elevation arrays are required as there are slots in the ring buffer.) Consequently, either a copy of the elevation array must be placed at each element of the ring buffer, or a separate ring of elevation arrays can be used to avoid unnecessary copying.

Since elevation tiles can only be rendered when the corresponding color tile is actually in memory, a node can only be expanded when the color tiles of all of its four children are

currently in memory. By locking in memory the color tiles for the top few levels of the quad tree, the test can be avoided for these levels. This not only makes the traversal faster, but it guarantees that some representation of the terrain can always be rendered, although perhaps at a coarser resolution than desired. The traversal of the truncated quad tree takes place just as described in the previous paragraph, except that a node is not expanded when the color tile of one of its children is not in memory. This can happen either while searching for the next elevation leaf node or while searching for the next color leaf node. In the former case, the node must immediately be treated as an elevation leaf node, but with no children. In the latter case, the node is simply treated as if it were a color leaf node.

The final stage is the basic rendering stage. This process constantly attempts to read the next available slot in the ring buffer. If the slot is the new frame marker, it then switches framebuffers, clears the back framebuffer where all rendering will take place, and renders any other objects required. Otherwise, it takes the elevation array and color tile pointers from the slot, binds the color tile into texture map memory, and converts the elevation data into quad-strips.

## 2.7 Maintaining a Constant Frame Rate

An important consideration in real-time rendering is maintaining as constant a frame rate as possible, at least while the user is moving. Since frame rate is a function of the complexity of the data being rendered (i.e., the number and size of facets, and the number and size of texture-maps, etc.), and since complexity is a function of the screen resolutions $r_z$ and $r_c$ (see Section 2.1), it should be possible to control frame rate by adjusting these two values.

A simple feedback mechanism based on the amount of time required to render the previous frame should be sufficient to maintain a constant frame rate. The simplest such mechanism is to decrease screen resolutions by a given percentage when the rendering time is greater than the desired rendering time, to increase it by a given percentage when the rendering time is smaller than the desired rendering time by a certain amount (to avoid oscillation), and to leave them unchanged otherwise. More sophisticated feedback mechanisms can be used if this proves to be inadequate.

The above assumed that a constant frame rate was desired. However, it may be the case that the resulting images are too coarse for a given frame rate because of the inadequate speed of the graphics engine. In this case, one can either attempt to trade off frame rate against image quality in order to meet some given optimality criterion, or one can simply let the user control the desired frame rate interactively.

There may also be situations where the highest quality rendering is desired, regardless of rendering time. This could be done, for example, by having the user interactively specify what screen resolutions to use when he/she has stopped moving for a given length of time. This approach has the advantage of maintaining constant frame rates while the user is moving over the terrain, yet providing high quality images when needed.

## 3.0 Pre-fetching Terrain Data

As indicated in the introduction, it is necessary to pre-fetch terrain data before it is actually required for rendering because of the inherent delays in retrieving the data from the database. In addition, the mechanism for pre-fetching data must be able to take into account the possibility of lost data.

For now, it is assumed that all of the elevation data and the first few top levels of the color data can fit in local memory. Consequently, these will be retrieved only once when the area of interest is chosen by the user. What needs to be pre-fetched, then, are color tiles in the lower levels of the color tile resolution pyramid.

In order pre-fetch the color tiles, it is necessary to predict what the user's viewpoint will be at some point $\Delta T$ from the current point in time. The simplest form of prediction is a linear one, which is $\Delta T$ times the rate of change of the position and orientation view parameters. A quadratic predictor based on the acceleration of the parameters will be used if this proves to be inadequate.

Given the inherent uncertainty in predicting a user's future viewpoint, it is necessary not only to pre-fetch those tiles within the predicted viewpoint, but also those in the surrounding area. This is accomplished by increasing the field of view as a function of $\Delta T$. A simple linear increase in the field of view with respect to $\Delta T$ should be adequate for this purpose.

Using an expanded field of view of the predicted viewpoint is a relatively simple mechanism that has the advantage that exactly the same code used for traversing the terrain quad tree for the current view can be used to create a truncated quad tree for the future view. The truncated quad tree is then traversed in a breadth-first fashion. For each node encountered, a test is made to see if the corresponding color tile is currently in memory (since the first few top levels are always in memory, the test is not necessary at these levels). If the tile is not currently in memory, the coordinates of the tile are placed at the bottom of the list of tiles to pre-fetch. This process is continued until either all of the nodes have been examined, or the maximum allowed number of tiles per pre-fetch, $N_p$, is reached. If the former occurs, and if the database retrieval protocol has a secondary list of "tiles to pre-fetch if there's time", the remaining tiles could be placed on this list. Note that this process assumes that the database server flushes all prior tile requests when it receives a new list.

The pre-fetch process described above, and illustrated in should be carried out at regular intervals in time (perhaps 5 times per second). The actual interval used may need to be determined empirically as a function of measured graphics and database retrieval performance.

Since it is not usually known *a priori* how many tiles per pre-fetch request that the database/network can support (because actual network bandwidth might change over time, for example), it is necessary to use a feedback mechanism to adjust $N_p$. For example, one

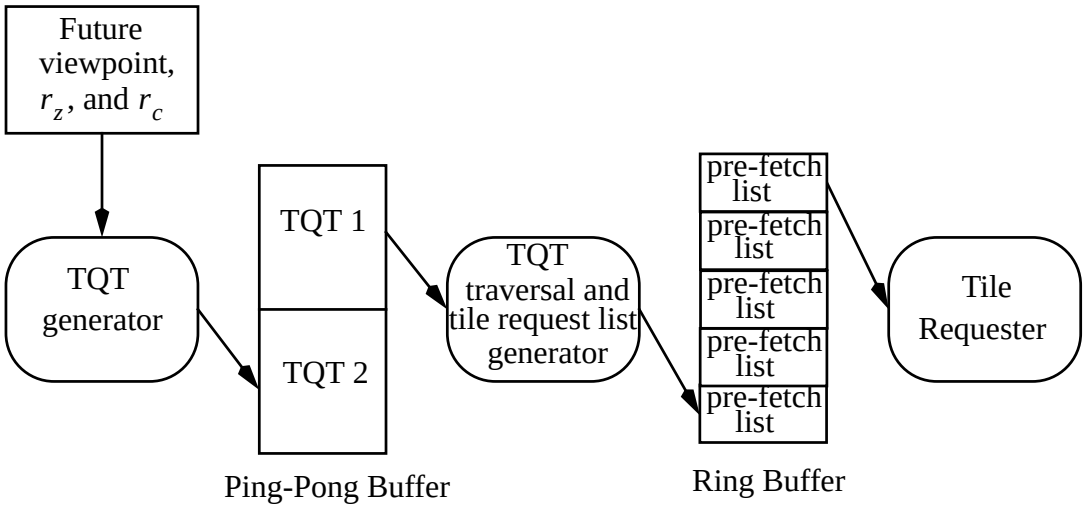


**FIGURE 7. TerraVision tile pre-fetching pipeline.**

might decrease $N_p$ until at least a certain ratio of received to requested tiles is achieved, increase $N_p$ when $N_p$ tiles were received, and leave $N_p$ unchanged otherwise.

Note that the second conditional above is *not* the same as saying "increase $N_p$ when all of the requested tiles were received." For example, if the user is moving quite slowly, only one tile per request might be initiated and received, even though $N_p$ might be, say, 10. Using the latter conditional, $N_p$ would be increased at every pre-fetch interval. Then, when the user starts moving more quickly, $N_p$ would have become quite large, possibly causing bottlenecks in the database retrieval. The former conditional, on the other hand, only increases $N_p$ when all of the tile requests have been satisfied, and the number of tile requests equals $N_p$. This way, $N_p$ only increases when there is evidence that the database retrieval can handle larger requests.

Since the truncated quad tree is traversed in a breadth-first fashion, the list of tiles to pre-fetch is naturally ordered from coarse to fine resolution tiles, or equivalently, from large area to small area tiles. Consequently, if the user stops moving, this process will first pre-fetch all of the visible tiles at a given resolution before pre-fetching tiles at the next highest resolution. Eventually, even if the database retrieval is quite slow, all of the visible tiles will reside in memory, and the desired resolution image (as specified by the desired screen resolutions) will be rendered. As the user begins to move, one expects that the number of new tiles required increases as the resolution increases. Consequently, for a given rate at which tiles can be requested, a maximum tile resolution will naturally arise as a function of the user's speed.

Finally, note that the above procedure automatically takes into account lost data. This is because all tiles that are potentially visible are tested to see if they're in memory at every pre-fetch cycle, independent of whether or not they were requested in the past, and because the database server is expected to flush all prior tile requests when it receives a

new request list. One possible disadvantage of this approach is that redundant requests can be made. For example, a tile might be en-route from the database server when the pre-fetch process tests to see if it is in memory. This tile would then be requested again, on the assumption that it had not been transmitted. This should happen rarely enough that it should not significantly effect the rate at which non-redundant tiles are transmitted.